Устранение рекурсии в полуинтерпретированных схемах программ

(Recursion Elimination in Semi-interpreted Program Schemata)

Николай Вячеславович Шилов (Университет Иннополис, shiloviis@mail.ru)

Доклад для Всероссийской научной конференции
«Математические основы информатики и информационно-коммуникационных систем» Тверской государственный университет, 3-8 декабря 2021 г.
(https://mfcsics.tversu.ru/)

Warming-up problem from IMO-2019

Part 0

Problem & Answer

- Let $\boldsymbol{Z}$ be the set of integers. Determine all functions $f\colon \boldsymbol{Z} \to \boldsymbol{Z}$ such that, for all integers a and b, $f(2a) + 2f(b) = f\big(f(a+b)\big)$.
- The problem can be solved using classical monadic recursion elimination technique known in Theoretical Computer Science since late 1960th: $f(x) = 2x + const$.

Problem via recursion elimination

- A classic example monadic recursion elimination by reduction to the tail recursion is a so-called John McCarthy function $M_{91}\colon \boldsymbol{N} \rightarrow \boldsymbol{N}$:

$$M_{91}(n) = if\ n > 100\ then\ (n - 10)\ else\ M_{91}\big(M_{91}(n + 11)\big).$$

- It was introduced by John McCarthy, studied by Zohar Manna, Amir Pnueli, Donald Knuth. It turns out that

$$M_{91}(n) = if\ n > 101\ then\ (n - 10)\ else\ 91.$$

Problem via recursion elimination

- A “key” idea elimination is a move from a monadic function $M_{91}: \boldsymbol{N} \rightarrow \boldsymbol{N}$ to a binary function $M2: \boldsymbol{N} \times \boldsymbol{N} \rightarrow \boldsymbol{N}$ such that for all $n, k \in \boldsymbol{N}$

$$M2(n, k) = (M_{91})^k(n)$$

where $(M_{91})^k(n)$ is k-time application of the function, i.e.:

- ○ $(M_{91})^k(n) = M_{91}(\dots M_{91}(n) \dots)$,
- ○ $M2(n, 0) = (M_{91})^0(n) = n$ for all $n \in \boldsymbol{N}$.

Levels of Recursion Elimination – interpreted and uninterpreted

Part 1

Recursive factorial

- Recursive program to compute the factorial function $F: \boldsymbol{N} \rightarrow \boldsymbol{N}$
  - $F(n) = if\ n = 0\ then\ 1\ esle\ n \cdot F(n-1)$ (in the standard notation),
  - $F(n) = if\ p(n)\ then\ c\ else\ f\left(n, F\big(g(n)\big)\right)$ (in a prefix notation),

  where “known” functions are
  - $p \equiv \big(\lambda\, x \in \boldsymbol{N}. (x = 0)\big) : \boldsymbol{N} \rightarrow Boolean,$
  - $c \equiv 1 :\rightarrow \boldsymbol{N}$ (i.e. a constant)
  - $f \equiv \big(\lambda\, x, y \in \boldsymbol{N}. (x \cdot y)\big) : \boldsymbol{N} \times \boldsymbol{N} \rightarrow \boldsymbol{N},$
  - $g \equiv \Big(\lambda\, x \in \boldsymbol{N}. \big(if\ x = 0\ then\ 0\ else (x-1)\big)\Big) : \boldsymbol{N} \rightarrow \boldsymbol{N}.$

Imperative factorial

Program 1

```
$VAR\ x, y: \mathbf{N};$
$y := 1;$
$while\ x \neq 0\ do$
    $y := x \cdot y;$
    $x := x - 1$
$od$
```

Program 2

```
$VAR\ x, y, z: \mathbf{N};$
$y := 1; z \coloneqq 1;$
$while\ z \leq x\ do$
    $y \coloneqq z \cdot y;$
    $z \coloneqq z + 1$
$od$
```

What if *known* functions are *uninterpreted*?

Recursive schemata with a single available (not specified) data type $\boldsymbol{T}$:

$$F(x) = if\ p(x)\ then\ c\ else\ f\left(x, F\left(g(x)\right)\right)$$

Standard scheme 1

```
$VAR\ x, y: \boldsymbol{T};$
$y := c;$
$while\ \neg p(x)\ do$
    $y := f(x, y);$
    $x := g(x)$
$od$
```

Standard scheme 2

```
$VAR\ x, y, z: \boldsymbol{T};$
$y := c; z := c;$
$while\ q(x, z)\ do$
    $y := f(z, y);$
    $z := h(z)$
$od$
```

Herbrand models and structures

- To demonstrate that no two of program schemata from the previous slide are equivalent, it is sufficient to consider *Herbrand models* (also called *free models*).
- The domain of a Herbrand model comprises all terms constructed from the available functional symbols and input variables (while the domain of the Herbrand structures comprise the ground terms exclusively).

Why the schemata aren't equivalent?

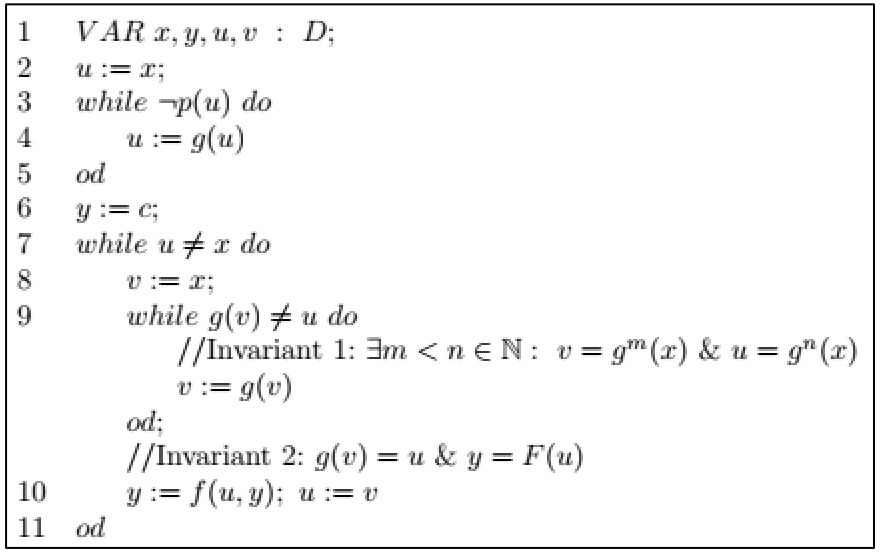
- Let us consider a Herbrand model such that
  - q is always $TRUE$,
  - $p\left(g\big(g(x)\big)\right)$ is $TRUE$ while p is $FALSE$ for all other terms.
- Then
  - $F(x) = f\left(x, F\big(g(x)\big)\right) = f\left(x, f\left(g(x), F\left(g\big(g(x)\big)\right)\right)\right) = f\big(x, f(g(x), c)\big)$,
  - the output value of y computed by scheme 1 is $f\big(g(x), f(x, c)\big)$,
  - while scheme 2 does not halt at all.

Translation of the recursive scheme to a standard scheme (with equality)

```
$VAR\ x, y, u, v : \boldsymbol{T};$
$u := x;$
$while\ \neg p(u)\ do$
    $u := g(u)$
$od$
$y := c;$
$while\ u \neq x\ do$
    $v := x;$
    $while\ g(v) \neq u\ do$
        $\underline{Inv.1:\ \exists m < n \in \boldsymbol{N} :\ v = g^m(x)\ \&\ u = g^n(x)}$
        $v := g(v)$
    $od;$
    $\underline{Inv.2:\ g(v) = u\ \&\ y = F(u)}$
    $y := f(u, y);\ u := v$
$od;$
$y := if\ p(x)\ then\ c\ else\ f(x, y)$
```

How to rid of the equality

- Finally, the equality used in lines 7 and 9 of the scheme is easy to eliminate because it may be implemented as call of the following *tail-recursive* function EQ (easy to implement by an iterative program:



```
1   $VAR\ x, y, u, v\ :\ D;$
2   $u := x;$
3   $while\ \neg p(u)\ do$
4       $u := g(u)$
5   $od$
6   $y := c;$
7   $while\ u \neq x\ do$
8       $v := x;$
9       $while\ g(v) \neq u\ do$
            $//\text{Invariant 1: } \exists m < n \in \mathbb{N} :\ v = g^m(x)\ \&\ u = g^n(x)$
            $v := g(v)$
        $od;$
        $//\text{Invariant 2: } g(v) = u\ \&\ y = F(u)$
10      $y := f(u, y);\ u := v$
11  $od$
```

$$EQ(a, b) =\ if\ p(a) \vee p(a)\ then\ p(a)\ \&\ p(b)\ else\ EQ\big(g(a), g(b)\big).$$

Translation of the recursive factorial to an iterative form

```
$VAR\ x, y, u, v : \mathbf{N};$
$u := x;$
$while\ u \neq 0\ do$
    $u := u - 1$
$od$
$y := 1;$
$while\ u \neq x\ do$
    $v := x;$
    $while\ (v - 1) \neq u\ do$
        $\underline{Inv.1: \exists m < n \in \mathbf{N} : v = x - m\ \&\ u = x - n}$
        $v := v - 1$
    $od;$
    $\underline{Inv.2:\ (v - 1) = u\ \&\ y = F(u)}$
    $y := u \cdot y;\ u := v$
$od;$
$y := if\ (x = 0)\ then\ 1\ else\ (x \cdot y)$
```

Extremely inefficient but semantic-independent

- Unfortunately, imperative factorial from the previous slide 10 is extremely inefficient – it runs in $O(n^2)$ time in contrast to both programs (1 and 2) from slide 4 that run in linear time $O(n)$.
- It worth to remark that Program 1 can be automatically constructed from the recursive factorial program using *co-recursion* and *tail-recursion.*
- This use of the co-recursion is semantic-dependent (since it is safe assuming commutativity of the function f), while our approach to recursion elimination is semantic-independent.

Co-recursion and Tail-recursion by example

- Recursive factorial $F(n) = if\ n = 0\ then\ 1\ esle\ n \cdot F(n-1)$ is not in the tail-form (because has next call inside some function).
- But it is equivalent to the following recursive program in the tail-form:

$$\begin{cases} F(n) = P(n, 1) \\ P(n, m) = if\ n = 0\ then\ m\ esle\ P\big((n-1), (n \cdot m)\big) \end{cases}.$$

- This program is in the tail-form because all calls are never inside other functions.
- Co-recursion is a “trick” that consists in converts result into another argument and use this argument in the recursion.

Teil-recursion elimination by example

- Tail-recursion $\begin{cases} F(n) = P(n,1) \\ P(n,m) = if\ n = 0\ then\ m\ esle\ P\big((n-1),(n \cdot m)\big) \end{cases}$

  is easy to eliminate (and compare with Program 1 from slide 4):

```
$\underline{start}$: $VAR\ x, y$: $\boldsymbol{N}\ goto\ \underline{2}$
$\underline{2}$: $y := 1\ goto\ \underline{3}$
$\underline{3}$: $if\ x = 0\ then\ goto\ \underline{stop}\ else\ goto\ \underline{4}$
$\underline{4}$: $y := x \cdot y\ goto\ \underline{5}$
$\underline{5}$: $x := x - 1\ goto\ \underline{3}$
stop
```

```
1. $VAR\ x, y: \boldsymbol{N}$;
2. $y := 1$;
3. $while\ x \neq 0\ do$
4.     $y := x \cdot y$;
5.     $x := x - 1$
6. $od$
```

Recursive and iterative Dynamic Programming

Part 2

Warming-up Dropping Bricks Problem

- Define stability of “bricks” (cell phones) by dropping them from a tower of H meters. How many times do you need to drop bricks, if you have just 2 bricks?
- $G(n) = if\ n = 0\ then\ 0\ else$
  $$1 + \min_{1 \leq k \leq n} \max\{(k-1), G(n-k)\}.$$

History of “Dynamic Programming”

- *Dynamic Programming* was introduced by Richard Bellman in the 1950s to tackle optimal planning problems.
- In 1950s the noun *programming* had nothing in common with more recent *computer programming* and meant *planning* (compare: *linear programming*).
- The adjective *dynamic* points out that *Dynamic Programming* is related to a *change of states* (compare – *dynamic logic*, *dynamic system*).

Bellman equation and optimality principle

- *Bellman equation* is a functional equality for the objective function that expresses the optimal solution at the *current* state in terms of the optimal solution at *next* (changed) states.
- It is conceptualized a so-called *Bellman Principle of Optimality*: an optimal plan (or program) should be optimal at every stage.

Descending (top-down) Dynamic Programming

- General pattern of Bellman equation may be formalised by the following *scheme of recursive descending Dynamic Programming*:

$$G(x) = if\ p(x)\ then\ f(x)\ else$$

$$\underbrace{g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in [1..n(x)]\right\}\right);}_{\text{the term is \textit{linear in each branch} w.r.t. the objective function G}}$$

Descending (top-down) Dynamic Programming – cont.

- In this scheme
    - $G: X \to Y$ is a symbol for the objective function,
    - p: $X \to Bool$ is a symbol for a known predicate,
    - $f: X \to Y$ is a symbol for a known function,
    - is a symbol for a known function with a variable (but finite) number of arguments,
    - all $h_i: X \times Z \to Y$, $i \in [1..n(x)]$ are symbols for known functions,
    - all $h_i: X \to X$, $i \in [1..n(x)]$ are symbols for known functions too.

More Examples:
Factorial, Fibonacci Numbers and Words

- $F(n) = if\ n = 0\ then\ 1\ else\ n \cdot F(n-1);$
- $Fib(n) = if\ 0 \leq n \leq 1\ then\ 1\ else\ Fib(n-2) + Fib(n-1);$
- $Wrd(n) = if\ n = 0\ then\ a$
  $else\ if\ n = 1\ then\ b$
  $else\ Wrd(n-2) \circ Wrd(n-1).$

Observations

- Factorial, Fibonacci Numbers and Words need static memory of a fixed size.
- Surprisingly, but Dropping Bricks Problem also needs just static memory of fix-size, since $G(n) = \arg\min k \in \boldsymbol{N}\colon \left(\frac{k(k+1)}{2} \geq n\right)$.

Problem under study

- It follows from Paterson M.S. and Hewitt C.T. paper *Comparative Schematology* (1970) that fix-size *static memory* is *not enough* for recursion elimination in Bellman equation.
- When one-time allocated
  - array (with integer indexes),
  - (fix-size) static memory

is sufficient to eliminate recursion in Bellman equation?

A Need of Dynamic Memory

- The following program scheme

  $$F(x) = if\ p(x)\ then\ x\ else\ f\left(F\big(g(x)\big), F\big(h(x)\big)\right)$$

  is not equivalent to any standard program scheme:

  for every $n > 0$

  there exists an Herbrand model T_n

  where any standard program scheme

  needs n variables to compute F.

Support of the Objective Function

- If $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in [1..n(x)]\right\}\right)$$
is defined for some value v, then it is possible to pre-compute the *support* $\mathrm{spp}(v)$, the set of all values that occur in the computation of $G(v)$:
$$\mathrm{spp}(x) = if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup \left(\bigcup_{i\in[1..n(x)]} \mathrm{spp}\left(t_i(x)\right)\right).$$
- Remark, that for every v, if $G(v)$ is defined, then $\mathrm{spp}(v)$ is finite (but not vice versa).

When an array suffices

- One-time allocated array with integer indexes suffices for computing $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in \left(1..n(x)\right)\right\}\right)$$

  if n is a constant and all t_i, $i \in \left(1..n(x)\right)$, are interpreted by commutative functions.

When static memory suffices

- Fix-size static memory suffice for computing

  $G(x) = if\ p(x)\ then\ f(x)\ else$

  $$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in \left(1..n(x)\right)\right\}\right)$$

  if $n(x) = n$ is a constant and there exists a known computable function t such that

  - $t_i = t^i$ for all $i \in [1..n]$,
  - $p(u)$ implies $p\left(t(u)\right)$ for all $u \in \mathrm{spp}(x)$.

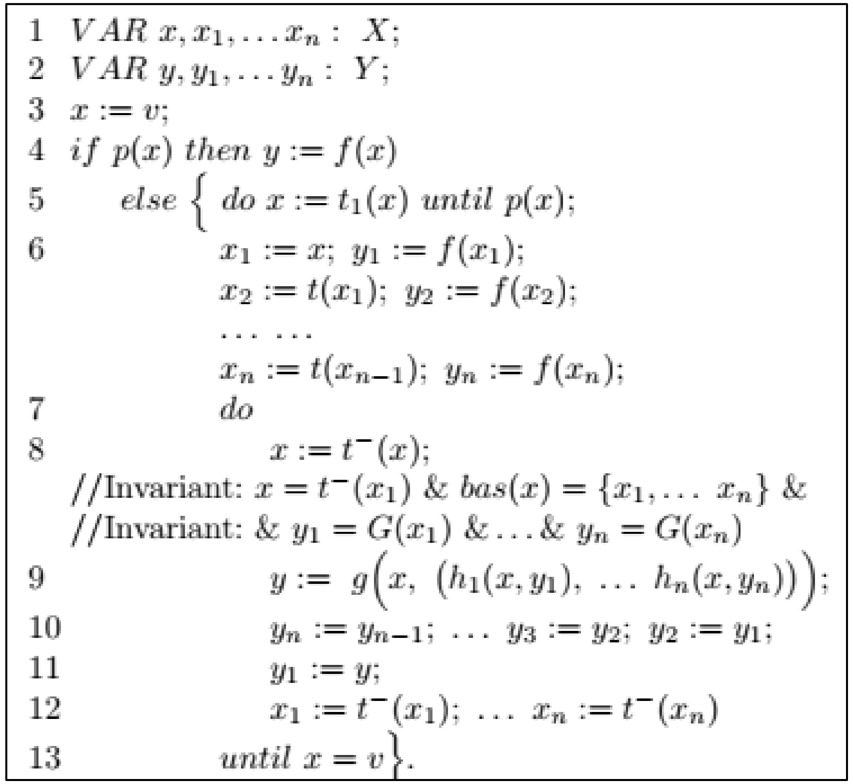
- Examples: Factorial, Fibonacci Numbers and Words.
- Counter-example: Paterson-Hewitt scheme.

Design outlines and proof comments

Proof comments

- Proof idea – very same as for factorial function in Part 1.
- Scheme’ design (with equality and invertible function t) is depicted to the right.

Design outlines



```
$VAR\ x, x_1, \dots x_n :\ X;$
$VAR\ y, y_1, \dots y_n :\ Y;$
$x := v;$
$if\ p(x)\ then\ y := f(x)$
    $else\ \{\ do\ x := t_1(x)\ until\ p(x);$
            $x_1 := x;\ y_1 := f(x_1);$
            $x_2 := t(x_1);\ y_2 := f(x_2);$
            $\dots\ \dots$
            $x_n := t(x_{n-1});\ y_n := f(x_n);$
            $do$
                $x := t^-(x);$
    $//\text{Invariant: } x = t^-(x_1)\ \&\ bas(x) = \{x_1, \dots\ x_n\}\ \&$
    $//\text{Invariant: } \&\ y_1 = G(x_1)\ \&\dots\&\ y_n = G(x_n)$
                $y :=\ g\Big(x,\ \big(h_1(x, y_1),\ \dots\ h_n(x, y_n)\big)\Big);$
                $y_n := y_{n-1};\ \dots\ y_3 := y_2;\ y_2 := y_1;$
                $y_1 := y;$
                $x_1 := t^-(x_1);\ \dots\ x_n := t^-(x_n)$
            $until\ x = v\}.$
```

References, concluding remarks, and topics for further research

Part 3

References

1. G. Berry. Bottom-up computation of recursive programs. RAIRO | Informatique Th´eorique et Applications (Theoretical Informatics and Applications, 10(3):47-82, 1976.
2. R. S. Bird. Zippy tabulations of recursive functions. In Proceedings of the 9th International Conference on Mathematics of Program Construction, MPC ’08, pages 92-109. Springer-Verlag, 2008.

References – cont.

3. J. Cowles and R. Gamboa. Contributions to the theory of tail recursive functions, 2004. Available at http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf.
4. D.E. Knuth. Textbook examples of recursion. arXiv:cs/9301113[cs.CC], 1991.
5. Y. A. Liu. Systematic Program Design: From Clarity to Efficiency. Cambridge University Press, 2013.

References – cont.

6. M.S. Paterson and C.T. Hewitt. Comperative schematology. In Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation, pages 119-127. Association for Computing Machinery, 1970.
7. N.V. Shilov. Etude on recursion elimination. Modeling and Analysis of Information Systems, 25(5):549-560, 2018.
8. N.V. Shilov, D. Danko Teaching Efficient Recursive Programming and Recursion Elimination Using Olympiads and Contests Problems. In Proc. of the workshop on Frontiers in Software Engineering Education (FISEE-2019), Lecture Notes in Computer Science, 2020, v.12271, p.246-264.

Concluding remarks

- A novelty of our study consists in use of templates (understood as semi-interpreted program schemata with symbol of a variable arity) and sematic sufficient conditions that allow recursive programs to be computed efficiently by iterative imperative programs (with either an associative or integer arrays or just with a finite fixed size static memory).

Further research topics

- All our sufficient conditions impose some constraints on interpretation of functional and predicate symbols. A very natural question s whether we can weaken these sufficient conditions?
- Computer-aided verification of the correctness of the translation of the descending dynamic programming template into iterative templates with arrays or fix-size static memory is a topic for further research.

IMO Grad Challenge

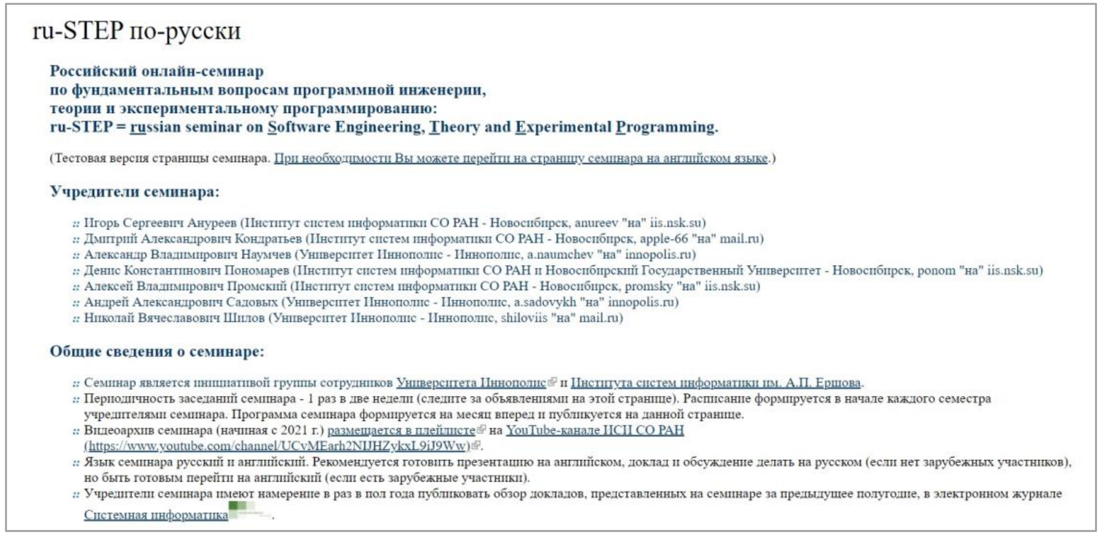
The International Mathematical Olympiad (IMO) is perhaps the most celebrated mental competition in the world and as such is among the ultimate grand challenges for Artificial Intelligence (AI).

The challenge: build an AI that can win a gold medal in the competition.

(https://imo-grand-challenge.github.io/)

Before saying Goodbye

- Добро пожаловать на Российский онлайн-семинар по фундаментальным вопросам программной инженерии, теории и экспериментальному программированию RuSTEP!



ru-STEP по-русски

Российский онлайн-семинар
по фундаментальным вопросам программной инженерии,
теории и экспериментальному программированию:
ru-STEP = russian seminar on Software Engineering, Theory and Experimental Programming.

(Тестовая версия страницы семинара. При необходимости Вы можете перейти на страницу семинара на английском языке.)

Учредители семинара:

:: Игорь Сергеевич Ануреев (Институт систем информатики СО РАН - Новосибирск, anureev "на" iis.nsk.su)
:: Дмитрий Александрович Кондратьев (Институт систем информатики СО РАН - Новосибирск, apple-66 "на" mail.ru)
:: Александр Владимирович Наумчев (Университет Иннополис - Иннополис, a.naumchev "на" innopolis.ru)
:: Денис Константинович Пономарев (Институт систем информатики СО РАН и Новосибирский Государственный Университет - Новосибирск, ponom "на" iis.nsk.su)
:: Алексей Владимирович Промский (Институт систем информатики СО РАН - Новосибирск, promsky "на" iis.nsk.su)
:: Андрей Александрович Садовых (Университет Иннополис - Иннополис, a.sadovykh "на" innopolis.ru)
:: Николай Вячеславович Шилов (Университет Иннополис - Иннополис, shiloviis "на" mail.ru)

Общие сведения о семинаре:

:: Семинар является инициативой группы сотрудников Университета Иннополис и Института систем информатики им. А.П. Ершова.
:: Периодичность заседаний семинара - 1 раз в две недели (следите за объявлениями на этой странице). Расписание формируется в начале каждого семестра учредителями семинара. Программа семинара формируется на месяц вперед и публикуется на данной странице.
:: Видеоархив семинара (начиная с 2021 г.) размещается в плейлисте на YouTube-канале ИСИ СО РАН (https://www.youtube.com/channel/UCvMEarh2NIJHZykxL9iJ9Ww).
:: Язык семинара русский и английский. Рекомендуется готовить презентацию на английском, доклад и обсуждение делать на русском (если нет зарубежных участников), но быть готовым перейти на английский (если есть зарубежные участники).
:: Учредители семинара имеют намерение в раз в пол года публиковать обзор докладов, представленных на семинаре за предыдущее полугодие, в электронном журнале Системная информатика.

- Russian seminar on Software Engineering, Theory and Experimental Programming (https://persons.iis.nsk.su/ru/ruSTEP).